MPJava: High-Performance Message Passing in Java using Java.nio

William Pugh and Jaime Spacco

University of Maryland, College Park, MD 20740, USA,
[pugh,jspacco]@cs.umd.edu

Abstract. We explore advances in Java Virtual Machine (JVM) technology along with new high performance I/O libraries in Java 1.4, and find that Java is increasingly an attractive platform for scientific cluster-based message passing codes.
We report that these new technologies allow a pure Java implementation of a cluster communication library that performs competitively with standard C-based MPI implementations.

1 Introduction

Previous efforts at Java-based message-passing frameworks have focused on making the functionality of the Message Passing Interface (MPI) [18] available in Java, either through native code wrappers to existing MPI libraries (mpiJava [3], JavaMPI [12]) or pure Java implementations (MPIJ [8]). Previous work showed that both pure Java and Java/native MPI hybrid approaches offered substantially worse performance than MPI applications written in C or Fortran with MPI bindings.

We have built Message Passing Java, or MPJava, a pure-Java message passing framework. We make extensive use of the `java.nio` package introduced in Java 1.4. Currently, our framework provides a subset of the functionality available in MPI. MPJava does not use the Java Native Interface (JNI). The JNI, while convenient and occasionally necessary, violates type safety, incurs a performance penalty due to additional data copies between the Java and C heaps, and prevents the JVM's Just-In Time (JIT) compiler from fully optimizing methods that make native calls.

MPJava offers promising results for the future of high performance message passing in pure Java. On a cluster of Linux workstations, MPJava provides performance that is competitive with LAM-MPI [9] for the Java Grande Forum's Ping-Pong and All-to-All microbenchmarks. Our framework also provides performance that is comparable to the Fortran/LAM-MPI implementation of a Conjugate Gradient benchmark taken from the NASA Advanced Supercomputing Parallel Benchmarks (NAS PB) benchmark suite.

2 Design and Implementation

We have designed MPJava as an MPI-like message passing library implemented in pure Java, making use of the improved I/O capabilities of the `java.nio` package. MPJava adheres to the Single Program Multiple Data (SPMD) model used by MPI. Each

MPJava instance knows how many total nodes are in use for the computation, as well as its own unique processor identification tag (PID). Using this information, the programmer can decide how to split up shared data. For example, if 10 nodes are being used for one MPJava computation, a shared array with 100 elements can store elements 0-9 on node 0, 10-19 on node 1, etc. Data can be exchanged between nodes using point-to-point `send()` and `recv()` operations, or with collective communications such as all-to-all broadcast. Distributing data in this manner and using communication routines is typical of MPI, OpenMP, and other parallel programming paradigms.

2.1 Functionality

The MPJava API provides point-to-point `send()` and `recv()` functions:

```
send( int peer, int offset, int len, double[] arr ) recv( int peer, int offset, int len, double[] arr )
```

These high-level functions abstract away the messy details related to TCP, allowing the user to focus on the application rather than the message passing details.

MPJava also provides a subset of the collective communication operations typically available to a message passing library such as MPI. For example, if an array with 100 elements is distributed between 10 nodes, an all-to-all broadcast routine can be used to recreate the entire array of 100 elements on each node:

```
alltoallBroadcast( double[] arr, int distribution )
```

The distribution parameter is a constant that tells MPJava how the data is distributed between nodes. The default setting we use is as follows: an array with n elements will be split between p nodes with each node holding n/p elements and the last node holding $n/p + n \bmod p$ elements. Other distribution patterns are possible, though we employ the simple default setting for our experiments.

2.2 Bootstrapping

MPJava provides a series of start-up scripts that read a list of hostnames, perform the necessary remote logins to each machine, and start MPJava processes on each machine with special arguments that allow each MPJava process to find the others. The result of the bootstrap process is a network of MPJava processes where each process has TCP connections to every other process in the network. These TCP connections are used by the nodes for point-to-point as well as collective communications.

2.3 Collective Communication Algorithms

We explored two different all-to-all broadcast algorithms: a multi-threaded concurrent algorithm in which all pairs of nodes exchange data in parallel, and a parallel prefix algorithm that only uses a single thread.

In the concurrent algorithm, each node has a separate send and receive thread, and the `select()` mechanism is used to multiplex communication to all the other processors.

In the parallel prefix implementation, data exchange proceeds in $log_2(\mathtt{n})$ rounds, sending 2^{r-1} pieces of data in each round, where r is the current round number. For example, if there were 16 total nodes, node 0 would broadcast according to the following schedule:

Example broadcast schedule for node 0 with 16 total nodes

round	partner	data
1	1	0
2	2	0,1
3	4	0-3
4	8	0-7

3 Introduction to java.nio

Java's New I/O APIs (`java.nio`), are defined in Java Specification Request (JSR) 51 [17]. These New I/O, or NIO, libraries were heavily influenced and address a number of issues exposed by the pioneering work of Matt Welsh et. al. on JAGUAR [21] and Chi-Chao Chang and Thorsten von Eicken on JAVIA [5].

3.1 Inefficiencies of java.io and java.net

The original `java.io` and `java.net` libraries available prior to JDK 1.4 perform well enough for client-server codes based on Remote Method Invocation (RMI) in a WAN environment. The performance of these libraries is not suitable, however, for high-performance communication in a LAN environment due to several key inefficiencies in their design:

- Under the `java.io` libraries, the process of converting between bytes and other primitive types (such as doubles) is inefficient. First, a native method is used that allows a double to be treated as a 64 bit long integer (The JNI is required because type coercions from double to long are not allowed under Java's strong type system). Next, bit-shifts and bit-masks are used to strip 8 bit segments from the 64 bit integer, then write these 8 bit segments into a byte array.
  `java.nio` buffers allow direct copies of doubles and other values to/from buffers, and also support bulk operations for copying between Java arrays and `java.nio` buffers.
- The `java.io` operations work out of an array of bytes allocated in the Java heap. Java cannot pass references to arrays allocated in the Java heap to system-level I/O operations, because objects in the Java heap can be moved by the garbage collector.
  Instead, another array must be allocated in the C heap and the data must be copied back and forth. Alternatively, to avoid this extra overhead, some JVM implementations "pin" the byte array in the Java heap during I/O operations.
  `java.nio` buffers can be allocated as `DirectBuffer`s, which are allocated in the C heap and therefore not subject to garbage collection. This allows I/O operations with no more copying than what is required by the operating system for any programming language.
- Prior to NIO, Java lacked a way for a single thread to poll multiple sockets, and the ability to make non-blocking I/O requests on a socket. The workaround solution of using a separate thread to poll each socket introduces unacceptable overhead for a high performance application, and simply does not scale well as the number of sockets increases.
  `java.nio` adds a unix-like `select()` mechanism in addition to non-blocking sockets.

3.2 java.nio for high-performance computing

MPJava demonstrates that Java can deliver performance competitive with MPI for message-passing applications. To maximize the performance of our framework we have made careful use of several `java.nio` features critical for high-performance computing: channels, `select()`, and buffers.

Channels such as `SocketChannel` are a new abstraction for TCP sockets that complement the `Socket` class available in `java.net`. The major differences between channels and sockets are that channels allow non-blocking I/O calls, can be polled and selected by calls to `java.nio`'s `select()` mechanism, and operate on `java.nio.ByteBuffer`s rather than byte arrays. In general, channels are more efficient than sockets, and their use, as well as the use of `select()`, is fairly simple, fulfills an obviou need, and is self-explanatory.

The use of `java.nio.Buffer`s, on the other hand, is slightly more complicated, and we have found that careful use of buffers is necessary to ensure maximal performance of MPJava. We detail some of our experiences with buffers below.

One useful new abstraction provided by NIO is a `Buffer`, which is a container for a primitive type. Buffers maintain a position, and provide relative `put()` and `get()` methods that operate on the element specified by the current position. In addition, buffers provide absolute `put(int index, byte val)` and `get(int index)` methods that operate on the element specified by the additional `index` parameter, as well as bulk `put()` and `get()` methods that transfer a range of elements between arrays or other buffers.

`ByteBuffer`s allocated as `DirectByteBuffer`s will use a backing store allocated from the C heap that is not subject to relocation by the garbage collector. The `DirectByteBuffer` the results can be directly passed as arguments to system level calls with no additional copying required by the JVM.

Because direct buffers are expensive to allocate and garbage collect, we pre-allocate all of the required buffers. The user sees a collective communication API call much like an MPI function; our framework handles behind-the-scenes any necessary copies of data between the user's arrays and our pre-allocated direct buffers.

For the best performance, it is important to ensure that all buffers are set to the native endianness of the hardware. Because Java is platform independent, it is possible to create buffers that use either big-endian or little-endian formats for storing multi-byte values. Furthermore, the default byte-order of all buffers is big-endian, *regardless of the native byte order of the machine where the JVM is executing*. To communicate among a set of heterogeneous platforms with mixed byte orders, one would need to perform some extra bookkeeping, and weather some performance overhead in the process. In our experience, this has never been an issues, as most clusters consist of a homogenous set of machines.

`ByteBuffer` provides an `asDoubleBuffer()` method that returns a `DoubleBuffer`, which is a "view" of the chunk of backing data that is shared with the `ByteBuffer`. Maintaining multiple "views" of the same piece of data is important for three reasons: First, while `ByteBuffer` supports operations to read or write other primitive types such as doubles or longs, each operation requires checks for alignment and endian-ness in addition to the bounds checks typical of Java. Next, `ByteBuffer` does not provide bulk transfer operations for other primitive types. Finally, all socket I/O calls require `ByteBuffer` parameters. NIO solves all of these issues with multiple views: `DoubleBuffer` provides bulk transfer methods for doubles that do not require checks for alignment and endian-ness. Furthermore, these transfers are visible to the `ByteBuffer`

"view" without the need for expensive conversions, since the `ByteBuffer` shares the same backing storage as the `DoubleBuffer`.

Maintaining two views of each buffer is cumbersome but manageable. We map each `DoubleBuffer` to its corresponding `ByteBuffer` with an `IdentityHashMap` and take care when changing the position of one of these buffers, as changes to the position of one buffer are not visible to other "views" of the same backing data. Furthermore, we are careful to prevent the overlap of simultaneous I/O calls on the same chunk of backing data, as the resulting race condition leads to nasty, unpredictable bugs.

The MPJava API calls in our framework take normal Java arrays as parameters. This approach requires that data be copied from arrays into buffers before the data can be passed to system-level OS calls. To avoid these extra copy operations, we initially implemented our framework with an eye towards performing all calculations directly in buffers. Not only does this strategy requires a more complicated syntax (buffers must be manipulated via `put()` and `get()` methods rather than the cleaner square bracket notation used with Java arrays), but the performance penalty for repeated `put()` and `get()` methods on a buffer is as much as an order of magnitude worse than similar code that uses Java arrays. It turns out that the cost of copying large amounts of data from arrays into buffers before every send (and from buffers to arrays after each receive) is less than the cost of the `put()` and `get()` methods required to perform computations in the buffers.

4 Performance Results

We conducted these experiments on a cluster of Pentium III 650 MHz machines with 768MB RAM, running Redhat Linux 7.3. They are connected by two channel-bonded 100 Mbps links through a Cisco 4000 switch capable of switching at maximum 45 million packets/s or 64 GB/s. We compared Fortran codes compiled with the g77-2.96 and linked with LAM-MPI 6.5.8 against MPJava compiled with JDK-1.4.2-b04 and mpiJava 1.2.3 linked with mpich 1.2.4.

We use mpich as the underlying MPI implementation for mpiJava because mpiJava supports mpich but not LAM. We chose LAM over mpich for our other experiments because LAM (designed for performance) delivers better performance than mpich (designed primarily for portability).

4.1 Ping-Pong

First we compare our MPJava framework with LAM-MPI and `java.io` for a ping-pong benchmark. The benchmark, based on the Java Grande Forum's ping-pong benchmark, measures the maximum sustainable throughput between two nodes by copying data from an array of doubles on one processor into an array of doubles on the other processor and back again. The results are given in Figure 1. The horizontal axis represents the number of doubles swapped between each pair of nodes. To avoid any performance anomalies occurring at the powers of two in the OS or networking stack, we adopt the Java Grande Forum's convention of using values that are similar to the powers of two. The vertical axis, labeled Mbits/s, shows bandwidth calculated as the total number of bits exchanged between a pair of nodes, divided by the total time for the send and receive operations. We only report results for the node that initiates the send first, followed by the receive, to ensure timing the entire round-trip transit time. Thus,

the maximum bandwidth for this benchmark is 100 Mbps, or half of the hardware maximum. We report the median of five runs because a single slow outlier can impact the mean value by a significant amount, especially for small message sizes where the overall transmission time is dominated by latency.

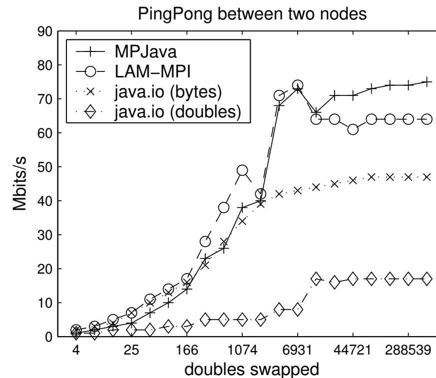

Fig. 1. Ping-Pong performance for MPJava, LAM-MPI, mpiJava, and java.io. Note that java.io (doubles) performs conversions between doubles and bytes, while java.io (bytes) does not

We used two different `java.io` implementations: `java.io` (doubles), which performs the necessary conversions from doubles to bytes and vice versa, and `java.io` (bytes), which sends an equivalent amount of data between byte arrays without conversions. The `java.io` (doubles) implementation highlights the tremendous overhead imposed by conversions under the old I/O model, while the results for `java.io` (bytes) represent an upper bound for performance of the old Java I/O model.

It is not surprising that our `java.nio`-enabled MPJava framework outperforms the `java.io` doubles implementation because the conversions are extremely inefficient. However, MPJava also outperforms the `java.io` (bytes) implementation for data sizes larger than about 2000 doubles. We surmise that this is due to inefficiencies in `java.io`'s buffering of data. Although both implementations need to copy data from the Java heap into the C heap, MPJava needs to copy data from a Java array into a pre-allocated direct buffer that does not need to be cleaned up, while the `java.io` (bytes) implementation needs to allocate and then clean-up space in the C heap. This may be an expensive operation on some JVMs.

The native LAM-MPI implementation provides better performance than MPJava for message sizes until about 1000 doubles, while MPJava provides superior performance for sizes larger than 7000 doubles.

The main contribution of this particular experiment is the empirical evidence we provide that Java is capable of delivering sustained data transfer rates competitive with available MPI implementation of this common microbenchmark.

4.2 All-to-All

The next microbenchmark we implemented was an all-to-all bandwidth utilization microbenchmark based on the Java Grande Forum's `JGFAlltoAllBench.java`.

The all-to-all microbenchmark measures bandwidth utilization in a more realistic manner than ping-pong. An all-to-all communication is necessary when a vector shared between many nodes needs to be distributed, with each node sending its portion to every other node. Thus, if there are `n` nodes and the vector has `v` total elements, each node must communicate its `v/n` elements to `n - 1` peers.

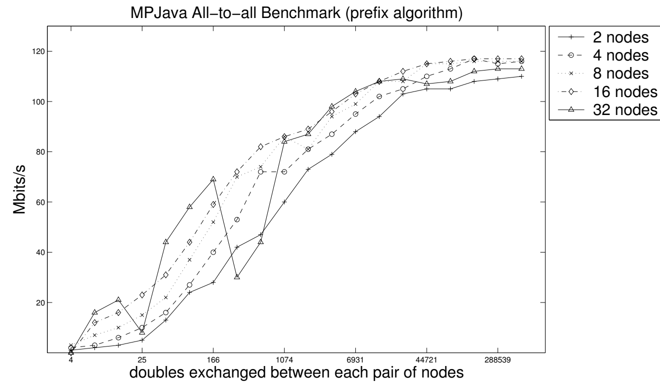

Fig. 2. All-To-All performance for MPJava, prefix algorithm

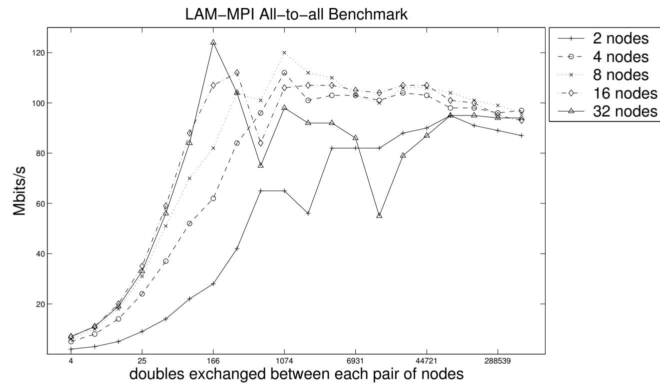

Fig. 3. All-To-All performance for LAM-MPI

Figure 2 represents the results of our framework using the parallel prefix algorithm, while Figure 4 shows the results for the concurrent algorithm. Figure 3 illustrates the performance of the same microbenchmark application written in C using the LAM-MPI

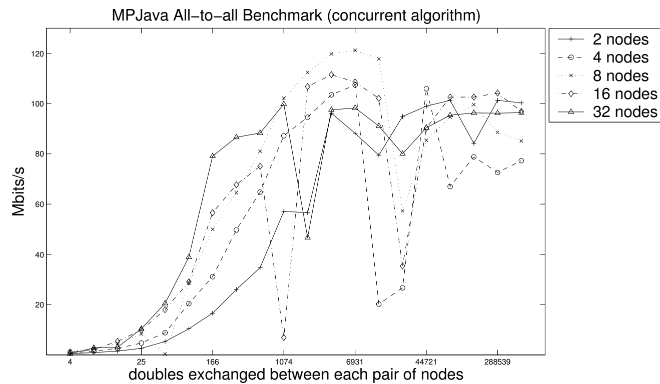

Fig. 4. All-To-All performance for MPJava, concurrent algorithm

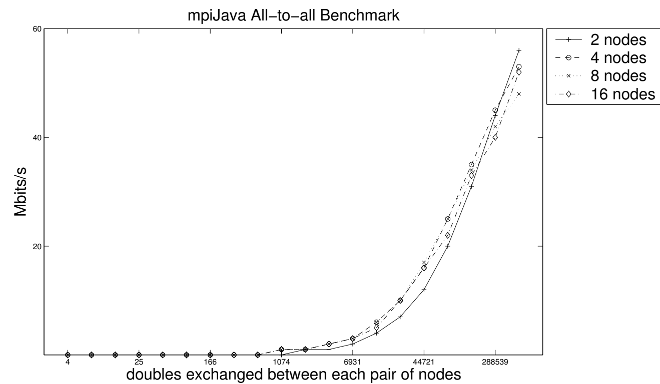

Fig. 5. All-To-All performance for mpiJava

library, and Figure 5 shows the results for mpiJava with bindings to mpich. Note that we do not chart mpiJava's performance for 32 nodes because the performance achieved was under 1 Mbps.

The values on the X-axis represent the number of doubles exchanged between each pair of nodes. A value `v` on the X-axis means that a total of `v * (n - 1)` bytes were transmitted, where `n` is the number of nodes used. The actual values selected for the X-axis are the same as those used in the ping-pong microbenchmark previously, for the same reason.

The Y-axis charts the performance in megabits/s (Mbps). We chose the median value of many runs because a single slow outlier can negatively impact the mean value, especially for small message sizes where overall runtimes are dominated by latency. Thus, the "dips" and other irregularities are repeatable. Note that Figures 2, 3 and 4 have the same scale on the Y-axis, and the theoretical hardware maximum for this experiment is 200 Mbps.

The MPJava concurrent broadcast algorithm occasionally outperforms the parallel prefix algorithm; however, the performance of the concurrent algorithm is not consis-

tent enough to be useful. We believe this is due at least in part to sub-optimal thread scheduling in the OS and/or JVM. In addition, we were not able to achieve true concurrency for this experiment because the machines we used for our experiments have only 1 CPU.

MPJava's parallel prefix algorithm outperformed the LAM-MPI implementation for large message sizes. We ascribe these differences to the difference in the broadcast algorithms. Parallel prefix has a predictable send/receive schedule, while LAM-MPI uses a naïve all-to-all algorithm that exchanges data between each pair of nodes.

The comparison with mpiJava is somewhat unfair because MPICH, the underlying native MPI library for mpiJava, gave substantially worse performance than LAM-MPI. However, the comparison does provide evidence of some of the performance hurdles that must be overcome for Java to gain acceptance as a viable platform for clustered scientific codes.

While it is possible that a C-based MPI implementation could use a more sophisticated broadcast strategy that outperforms our current implementation, there is no reason why that strategy could not be incorporated into a `java.nio` implementation that would achieve similar performance.

4.3 CG

Our final performance results are for the NAS PB Conjugate Gradient (CG) benchmark [19]. The CG benchmark provides a more realistic evaluation of the suitability of Java for high performance scientific computation because it contains significant floating point arithmetic.

The CG algorithm uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzero values.

The kernel of the CG algorithm consists of a multiplication of the sparse matrix `A` with a vector `p` followed by two reductions of a double, then a broadcast of the vector `p` before the next iteration. These four core operations comprise over 80% of the runtime of the calculation. This kernel iterates 25 times, and is called by the CG benchmark 75 times to approximate a solution with the desired precision.

We have evaluated the CG benchmark for the Class B and Class C sizes.

Class	rows of `A`	total nonzeroes in `A`	avg. nonzeroes/row
B	75,000	13,708,072	183
C	150,000	36,121,058	241

The data used by the CG benchmark is stored in Compressed Row Storage (CRS) format. The naïve way to parallelize this algorithm is to divide the `m` rows of the `A` matrix between `n` nodes when performing the $A.p$ matrix-vector multiplication, then use an all-to-all broadcast to recreate the entire `p` vector on each node. We implemented this approach in Fortran with MPI and also MPJava, and provide results for this approach in Figure 6.

Because g77 does not always adequately optimize code, we also ran the NAS CG benchmark using pgf90, the Portland Compiler Group's optimizing Fortran compiler. The performance was nearly identical to the g77 results. It is likely that even a sophisticated compiler cannot optimize in the face of the extra layer of indirection required by the CRS storage format for the sparse matrix `A`.

The NAS PB implementation of CG performs a clever two-dimensional decomposition of the sparse matrix `A` that replaces the all-to-all broadcasts with reductions

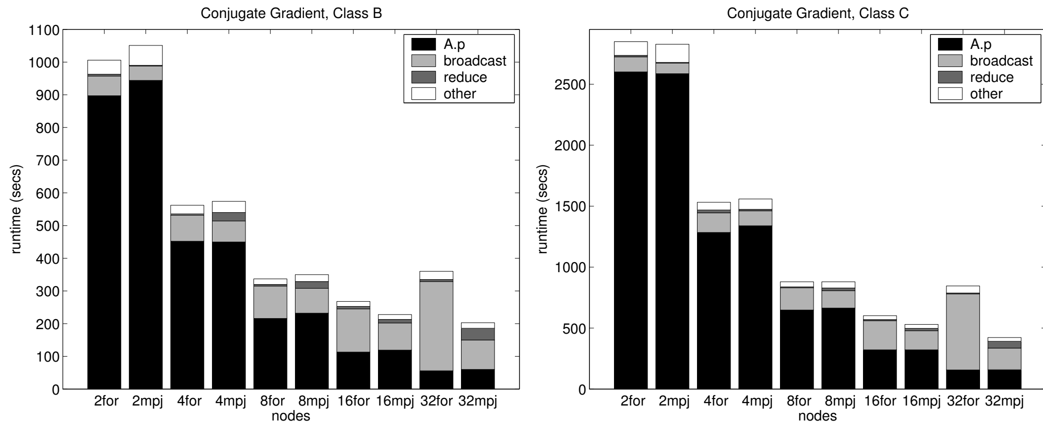

Fig. 6. Conjugate Gradient, Class B: MPJava (mpj), Simple Fortran (for). Note that for each pair of stacked bar charts, MPJava is the leftmost, simple Fortran is the rightmost

across rows of the decomposed matrix. The resulting communication pattern can be implemented with only `send()` and `recv()` primitives, and is more efficient than using collective communications. We implemented the more sophisticated decomposition algorithm used by the NAS CG implementation in MPJava, and provide results in Figure 7.

We instrumented the codes to time the three major contributors to the run-time of the computation: the multiplication of the sparse matrix `A` with the vector `p`, the all-to-all broadcast of the vector `p`, and the two reductions required in the inner loop of the CG kernel. All four versions of the code perform the same number of floating point operations. We report results for four versions: naïve Fortran (Fortran), the naïve MPJava (MPJava), Fortran with the 2D decomposition (Fortran 2D), and MPJava with the 2D decomposition (MPJava 2D). These results are in Table 8 for the Class B problem size, and Table 9 for the Class C problem size.

The results of the naïve algorithm presented in Figure 6 show that MPJava is capable of delivering performance that is very competitive with popular, freely-available, widely-deployed Fortran and MPI technology. The poor performance observable at 32 nodes for the Fortran code reflects the fact that LAM-MPI's all-to-all collective communication primitive does not scale well. These results highlight the importance of choosing the appropriate collective communication algorithm for the characteristics of the codes being executed and the hardware configuration employed.

The results of the 2D decomposition algorithm presented in Figure 7 also show MPJava to be competitive with Fortran and MPI. Although the MPJava performance is slightly worse, it is within 10% of the Fortran/MPI results. Popular wisdom suggests that Java performs at least a factor of 2 slower than Fortran. While there is much work left to do in the field of high-performance Java computing, we hope that our results help bolster Java's case as a viable platform for scientific computing.

The results of this benchmark suggest that MPJava is capable of delivering performance comparable to or in excess of the performance achievable by native MPI/C applications.

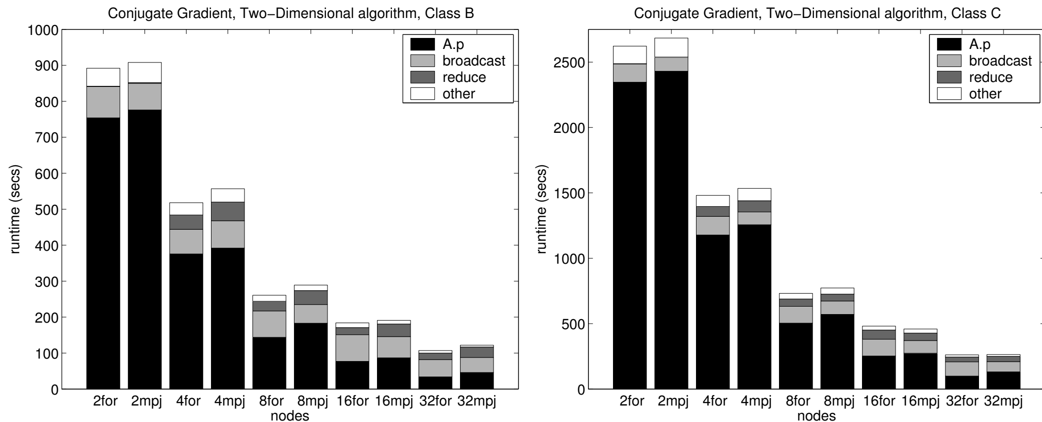

Fig. 7. Conjugate Gradient, Class B: MPJava (mpj), Original NAS Fortran (for). Note that for each pair of stacked bar charts, MPJava is the leftmost, NAS Fortran is the rightmost

In addition, this benchmark provides promising results for the current state of Java Virtual Machine (JVM) technologies. The results of the `A.p` sparse matrix-vector multiplications are nearly identical between the Simple Fortran and simple MPJava versions, and MPJava performs within 0% of Fortran for 2D versions. The only optimization we performed on the `A.p` sparse matrix-vector multiplication code was unrolling the loop by a factor of 8, which accounted for an improvement of about 17% for the Simple MPJava implementation. We assume that the Fortran compilers already perform this optimization, as loop unrolling by hand had no effect on Fortran code compiled with either g77 or pgf90.

5 Related Work

There is a large body of work dealing with message-passing in Java. Previous approaches can be loosely divided into two categories: Java/native hybrids, and `java.io` approaches.

JavaMPI [12] and mpiJava [3] are two efforts to provide native method wrappers to existing MPI libraries. The resulting programming style of JavaMPI is more complicated, and mpiJava is generally better supported. Both approaches provide the Java programmer access to the complete functionality of a well-supported MPI library such as MPICH [14].

This hybrid approach, while simple, does have a number of limitations. First, mpiJava relies on proper installation of an additional library. Next, the overhead of the Java Native Interface (JNI) imposes a performance penalty on native code which will likely make the performance of an application worse than if it were directly implemented in C with MPI bindings. Furthermore, the JIT compiler must make maximally conservative assumptions in the presence of native code and may miss potential optimizations.

language	nodes	total runtime	A.p	broadcast	reductions	other
Fortran	2	1006	897	60	6	43
MPJ	2	1052	944	44	2	61
Fortran 2D	2	892	754	87	1	50
MPJ 2D	2	907	776	74	2	56
Fortran	4	561	452	80	4	26
MPJ	4	574	450	64	26	34
Fortran 2D	4	519	376	68	40	34
MPJ 2D	4	556	392	76	52	37
Fortran	8	337	216	99	5	17
MPJ	8	351	232	76	21	21
Fortran 2D	8	261	144	73	27	17
MPJ 2D	8	289	183	52	39	15
Fortran	16	268	113	132	8	15
MPJ	16	228	119	83	11	15
Fortran 2D	16	184	77	74	20	13
MPJ 2D	16	190	87	59	35	10
Fortran	32	361	56	273	6	25
MPJ	32	203	60	90	36	17
Fortran 2D	32	107	34	48	18	7
MPJ 2D	32	122	46	42	29	5

Fig. 8. Raw results for Class B Conjugate Gradient benchmark. All times reported are in seconds.

language	nodes	total runtime	A.p	broadcast	reductions	other
Fortran	2	2849	2601	124	12	112
MPJ	2	2827	2587	86	6	149
Fortran 2D	2	2622	2346	140	2	134
MPJ 2D	2	2684	2430	107	2	145
Fortran	4	1532	1285	160	24	63
MPJ	4	1558	1339	123	12	85
Fortran 2D	4	1482	1178	142	76	85
MPJ 2D	4	1534	1256	98	86	94
Fortran	8	881	648	183	8	41
MPJ	8	879	664	143	22	51
Fortran 2D	8	732	504	128	57	43
MPJ 2D	8	774	571	102	53	47
Fortran	16	602	322	238	10	32
MPJ	16	531	322	156	20	32
Fortran 2D	16	482	253	128	70	30
MPJ 2D	16	459	274	97	57	31
Fortran	32	846	157	623	8	58
MPJ	32	422	159	177	56	31
Fortran 2D	32	260	99	109	37	16
MPJ 2D	32	264	132	77	41	14

Fig. 9. Raw results for Class C Conjugate Gradient benchmark. All times reported are in seconds.

Most `java.io` implementations are based on the proposed MPJ standard of Carpenter et. al. [4]. However, there is no official set of MPI bindings for Java, so each implementation will have its own particular advantages and disadvantages.

MPIJ, part of the Distributed Object Groups Metacomputing Architecture (DOGMA) project at BYU [8], is a pure-Java implementation of a large subset of MPI features. Their implementation is based on the proposed MPI bindings of Carpenter et. al. [4]. The MPIJ codebase was not available for public download at the time of publication. Steve Morin provides an excellent overview of MPIJ's design here [13]. We were unable to find any published results of the performance of MPIJ.

The Manta project [11] supports several interesting flavors of message-passing codes in Java, including Collective Communication Java (CCJ) [15], Group Method invocation (GMI) [10], and Ibis [20]. CCJ is an RMI-based collective communication library written entirely in Java. It provides many features, but does not provide the all-to-all broadcast necessary for many scientific codes such as Conjugate Gradient. GMI is a generalization of Java RMI in which methods can be invoked on a single object or on a group of objects, and results can be discarded, returned normally, or combined into

a single result. This work is extremely interesting from a high-level programmatic perspective, as its fully orthogonal group-based design potentially allows programs that break from the SPMD model so dominant to MPI codes. Ibis harnesses many of the techniques and infrastructures developed through CCJ and GMI to provide a flexible GRID programming environment.

MPI Soft Tech Inc. announced a commercial endeavor called JMPI, an effort to provide MPI functionality in Java. However, they have yet to deliver a product; all we have are their design goals [2].

JCluster [7] is a message-passing library that provides PVM and MPI-like functionality in Java. The library uses threads and UDP for improved performance, but does not utilize `java.nio`. The communications are thus subject to the inefficiencies of the older `java.io` package. At the time of publication, an alpha version of the library was available for Windows but did not work properly under Linux.

JPVM [6] is a port of PVM to Java, with syntactic and semantic modifications better suited to Java's capabilities and programming style. The port is elegant, full-featured, and provides additional novel features not available to PVM implementations in C or Fortran. However, the lackluster performance of JPVM, due in large part to the older io libraries, has proved a limiting factor to its wide adoption.

KARmi [16] presents a native-code mechanism for the serialization of primitive types in Java. While extremely efficient, native serialization of primitive types into byte arrays violates type safety, and cannot benefit from `java.nio` SocketChannels.

Titanium [22] is a dialect of Java that provides new features useful for high-performance computation in Java, such as immutable classes, multidimensional arrays, and zone-based memory management. Titanium's backend produces C code with MPI calls. Therefore the performance is unlikely to outperform native MPI/C applications, and could be substantially worse.

Al-Jaroodi et. al. provide a very useful overview of the state of distributed Java endeavors here [1].

Much work has been done on GRID computing. Our work does not directly deal with issues important to the GRID environment, such as adaptive dynamic scheduling or automatic parallelism. Rather, we focus of developing an efficient set of communication primitives that any GRID-aware library can be built on top of.

6 Conclusion

We have built a pure Java message-passing framework using NIO. We demonstrate that a message passing framework that harnesses the high-performance communication capabilities of NIO can deliver performance competitive with native MPI codes.

We also provide empirical evidence that current Java virtual machines can produce code competitive with static Fortran compilers for scientific applications rich in floating point arithmetic.

7 Future Work

Though MPI supports asynchronous messages, it typically does so without the benefit of threads, and in a cumbersome way for the programmer. We have a modified version of our framework that provides the abstraction of asynchronous pipes. This is accomplished through separate send and receive threads that make callbacks to user-defined

functions. We would like to evaluate the performance of our asynchronous message-passing framework for problems that do not easily fit into an SPMD model, such as distributed work-stealing and work-sharing.

Clusters are increasingly composed of interconnected SMPs. It is typically not useful to schedule multiple MPI tasks on an SMP node, as the additional processes will fight over shared resources such as bandwidth and memory. A Java framework that supports the interleaving of computation and communication through send, receive and compute threads can better utilize extra processors because the JVM is free to schedule its threads on all available processors. We have developed a threaded version of our MPJava framework that maintains a send, receive and computation thread. In the CG algorithm, since each node only needs the entire `p` vector for the `A.p` portion of any iteration, and the broadcast and matrix-vector multiply step are both significant contributors to the total runtime, we use threads to interleave the communication and computation of these steps. Our preliminary results were worse than the single-threaded results, most likely due to poor scheduling of threads by the OS and the JVM. The notion of interleaving computation and communication, especially on an SMP, is still very appealing, and requires more study. Multi-threading is an area where a pure-Java framework can offer substantial advantages over MPI-based codes, as many MPI implementations are not fully thread-safe.

Although interest in an MPI-like Java library was high several years ago, interest seems to have waned, perhaps due to the horrible performance reported for previous implementations. Now that NIO enable high-performance communication, it is time to reassess the interest in MPI-Java.

Finally, we would like to investigate the high-level question of whether a high-performance message-passing framework in Java should target MPI, or should adhere to its own standard.

8 Acknowledgments

This work was supported by the NSA, the NSF and DARPA.

References

1. Al-Jaroodi, Mohamed, Jiang, and Swanson. A Comparative Study of Parallel and Distributed Java Projects. In *IPDPS Workshop on Java for Parallel and Distributed Computing*, 2002.
2. JMPI, http://www.mpi-softtech.com/publications/jmpi121797.html.
3. M. Baker, B. Carpenter, S. Ko, and X. Li. mpiJava: A Java interface to MPI, 1998.
4. B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for java. *Concurrency - Practice and Experience*, 12(11):1019–1038, 2000.
5. C.-C. Chang and T. von Eicken. Interfacing Java with the Virtual Interface Architecture. *ACM Java Grande*, 1999.
6. A. Ferrari. JPVM: network parallel computing in Java. *Concurrency: Practice and Experience*, 10(11–13):985–992, 1998.
7. JCluster, http://vip.6to23.com/jcluster/, 2002.
8. G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Management Architecture. In *Concurrency: Practice and Experience*, pages ??–??, 1998.
9. LAM (Local Area Multicomputer), http://www.lam-mpi.org, 2002.

10. J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Languages, Compilers, and Runtime Systems*, pages 1–6, 2002.
11. Manta: Fast Parallel Java, http://www.cs.vu.nl/manta/.
12. S. Mintchev. Writing programs in javampi. Technical Report MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, October 1997.
13. S. R. Morin, I. Koren, and C. M. Krishna. Jmpi: Implementing The Message Passing Interface Standard In Java. In *IPDPS Workshop on Java for Parallel and Distributed Computing*, April 2002.
14. MPICH-A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich/, 2002.
15. A. Nelisse, T. Kielman, H. Bal, and J. Maassen. Object Based Collective Communication in Java. *Joint ACM Java Grande - ISCOPE Conference*, 2001.
16. C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Java Grande*, pages 152–159, 1999.
17. JSR 51 - New I/O APIs for the JavaTM Platform, http://www.jcp.org/jsr/detail/51.jsp, 2002.
18. The Message Passing Interface Standard.
19. The NAS Parallel Benchmarks, http://www.nas.nasa.gov/nas/npb/.
20. R. V. van Nieuwpoort, A. Nelisse, T. Kielman, H. Bal, and J. Maassen. Ibis: an Efficient Java-bsed Grid Programming Environment. *Joint ACM Java Grande - ISCOPE Conference*, 2002.
21. M. Welsh and D. Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.
22. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.